# 2. The Necessary Theory Part 2/3

# Table of Contents


- Introduction
- Functions
- Process vs Thread [1]
- Function Calling ([2] - Vol. 1, Chapter 6)
- Stack Frames ([2] - Vol. 1, Chapter 6)
    - Setting up the stack frame - a graphical example
- Calling Conventions [3]
- Reading EIP - A simple trick
- Conclusions
- References


# Introduction

So here we are in the second chapter, which is also dedicated to theoretical knowledge necessary for this course.

What you need to keep in mind during this course is that 'theoretical' doesn't actually mean that you might need it... or not.

In fact, the theory discussed in the first three chapters covers all the fundamental knowledge and concepts that you will need not just for this course and its technical assignments, but for the rest of your time as a reverser.

During this chapter we'll discuss functions, the difference between a thread and a process, the stack, how it works, what it is needed for, and how stack frames are created.

Finally, we will be discussing the most common calling conventions.

# Functions

All modern software is actually a set of smaller sub-programs or functions. Each function is dedicated to a specific operation and either returns a value to the function that called it, or not.

Likewise, a function might require a parameter or a set of parameters as input. of course, this depends on the needs of the algorithm that is executed inside that function.

There are several reasons behind the process of breaking-down a program into several sub-programs. First of all, it makes the process of understanding and maintaining the code much

easier. You might think when programming that this is your code so you wouldn't have to break it down, but when you need to come back to it after a year or more in order to change something or repair a bug, it becomes terribly time-consuming to scan through lots of lines of code in order to locate the part that you need.

Also, in the case of a bug, it's much easier to identify if all the important tasks of the program are placed inside dedicated functions. It simply makes it easier to know where to look when the time comes to dive back in.

In addition, you often need the same function more than once in the same program. So, instead of repeating the same piece of code over and over again, you can just place it in a function and just call that function whenever it is needed.

Of course, a function can call more sub-functions (sometimes called sub-routines), in case it's necessary to break down a complex algorithm into smaller parts for the same reasons we just discussed.

Having said that, it becomes obvious that when a function is called, something changes in the execution flow of the program. instructions in memory are executed one after another, and as we noted in the previous chapter, the instruction pointer (IP) always points to the memory location of the next instruction to be executed.

So, when the program needs to call a function and transfer execution flow to that function, the program needs a way to remember where execution left off. This is required in order for the program to be able to continue once the function is complete: to return and continue executing the next instruction after the one that transferred the execution to the function.

Also, that function may need a set of parameters in order to operate properly, and in addition to this we might want to preserve some register values in order to be able to retrieve them once we exit the function.

This is where the procedure stack comes into play - also know as "the stack".

The stack is a reserved memory area inside a specific segment called the stack segment, which is identified by the segment selector inside the SS register. The term "reserved" is used in this case in order to indicate its special use during program execution

# Process vs Thread [1]

Each thread executing inside a process has its own stack. This means that if a process has five threads, then there is a stack assigned for each one - five stacks in total. At this point this might not make sense, but it will.

In order to distinguish the concept of a process from a thread, think about a process as the container and a thread as something inside the container.

When you execute an application, Windows OS creates the process which is identified by a virtual address space in memory where all the executable and non-executable modules/resources needed are mapped. The OS also creates at least one thread that's ready for execution.

In fact, every process has its own virtual address space, but multiple threads inside the same process share the same virtual address space - the one of the process they belong to.

On the other hand, the term thread can indicate the time-slice assigned by the processor to that specific process in order to execute a portion of instructions inside it.

In other words, in a single-processor system, only one thread can be executed at a time, but on a multi-processor or multi-core system it is possible to have as many threads running simultaneously as the number of cores.

Of course, all this depends on the way the operating system uses the available resources. In other words, it is it he operating system that assigns the various threads to the available processors, in order to actually take advantage of the presence of more than one processors.

Remember a few years ago when only single-core processors were avaialble, and you could still run many programs at once? In reality, the processor was assigning a time-slice for execution on each thread at the time, but due to the speed with which this was done, you thought that all applications were running at the same time.

Actually, the same thing happens today. You might have  a quad-core processor, but you aren't limited to four processes running at the same time. The illusion of simultaneous execution of multiple application at the same time is called multitasking. Having multiple threads that actually run in parallel is called multithreading.

# Function Calling ([2] - Vol. 1, Chapter 6)

As we already mentioned, some functions might require one or more parameters. A parameter is a specific type of variable. It can be a signed or unsigned integer, a pointer to an array, a string, a single char, etc.

When the function requires one or more parameters, these parameters are placed on the stack before entering that function.

The 2 main instructions used for inserting and extracting values from the stack are the PUSH and POP instructions.

The PUSH instruction places data on top of the stack and the POP instruction removes that value from the top of the stack (TOS). Due to the way the stack operates, this is considered a LIFO Data Structure (Last In - First Out).

What actually happens is that when an item is pushed on the stack, the ESP register, which always points to the TOS, is decremented in order to point to the new item placed on the TOS.

During this course we will assume that the width of the stack is 32bits.

An example:

*Let's say that you have 4 books, one on top of the other, and each book occupies 32 bits of data. The book at the bottom (where EBP - the base pointer - points) occupies the highest position (address), assuming that we start counting from the top of the pile.*

*The ESP always points to the last book placed on that pile, so it will always point to the book that is on top of the rest.*

*Now, you want to add (PUSH) another book to the pile, and since the ESP register must always point to the top of the pile, it should be decremented by 32 bits (4 bytes) in order to point to the last book added.*

*On the other hand, if we want to remove a book from the top of that pile, the ESP pointer must be incremented by the same value in order to point to the next book after the one we just removed.*

In reality, the stack during execution of the program is divided in several stack-frames, which we will explain in detail in the next section of this chapter.

So, once the parameters - if required by the function - are pushed on the stack, the program needs to redirect the EIP to the entry point of that function and at the same time, it needs to keep track of the address of the next instruction to execute on returning from that function.

The instruction responsible for this is the CALL instruction.

When a CALL instruction is executed, what happens is that the processor pushes the return address (called the 'return-instruction pointer') onto the stack and then loads the address of the entry point of that function in the EIP register in order to start the execution of that function.

The corresponding instruction for exiting from a function is the RET instruction.

When a RET instruction is executed, the processor pops the address from TOS to the EIP register and resumes execution from the next instruction located after the CALL that brought us inside the function in the first place.

In some cases, the RET instruction can be used with an argument, for example RET 4. In this case the ESP pointer will not just be incremented by 32 bits (4 bytes) during the popping of the value in TOS, but also add the value of the arguments (in bytes).

This is used in __stdcall and __fastcall calling conventions in order to clean up the stack from the parameters pushed for that function (see 2.6 Calling Conventions).

# Stack Frames ([2] - Vol. 1, Chapter 6)

As mentioned at the beginning of this chapter, applications are comprised of a set of functions, and each of these functions may CALL other subroutines, and in addition these functions may have some local variables and other parameters needed.

All this makes clear that there must be  away for the program in execution to 'remember' the return address each time the execution flow enters a function and of course each function needs to have its own memory area where it can store all of the above (return address, local variables, etc.)

This is done by splitting the memory assigned for the stack into stack frames. Each frame holds the information just mentioned for each function.

Every time we enter a function, a stack frame for this function is created through the function prologue. A function prologue is a set of instructions for which the purpose is to allocate the necessary memory area for that specific function inside the stack, and it is the first thing done every time we enter a function.

Very often, immediately after the function prologue we notice that some of the general purpose registers are pushed onto the stack. This happens because we might want to use these registers during the execution of the function, but we will still need their initial values once this function is completed, so its stack frame is used as a temporary storage area.

However, keep in mind that the memory size allocated by the OS for the stack, used by the thread under execution, is fixed. This means that we cannot just keep pushing values on it indefinitely, and this is also one of the reasons why we always clean up the stack at the end of a function.

Also, once the function is complete, we need to re-balance the stack; in other words, 'free' the memory allocated for this stack frame, and restore the values of ESP and EB registers. This is done by the function epilogue, which is, again, a set of instructions that we'll see in a while.

In case the initial values of other general purpose registers were saved after the function prologue, then these have to be popped out of the stack into their respective registers before the function epilogue occurs. Remember, last-in first-out.

Code Snippet: Function Prologue/Epilogue Example:

```
push ebp // push the base pointer to the stack to save it
mov ebp,esp // move to the base pointer the value of the stack pointer
sub esp,10h // allocate 10h (16 decimal) bytes of space for the current stack
frame



push eax // we might want to save the values of other general purpose
registers
push ebx // same as above



add eax,ebx // start of function body
xor ebx,eax
sub ebx,eax // end of function body
pop ebx // restore EBX
pop eax // restore EAX



mov esp,ebp // start function epilogue (free memory)
pop ebp // restore base pointer
ret // exit function
```

S

# Setting up the stack frame - a graphical example

During this example we assume that we just entered a function which receives 2 parameters, which are pushed onto the stack before entering the function. Remember, the stack's width is 32-bit, which means that each one of its units occupies 4 bytes of memory.

Figure 2.1 demonstrates the current state of the stack before the execution of the function prologue.

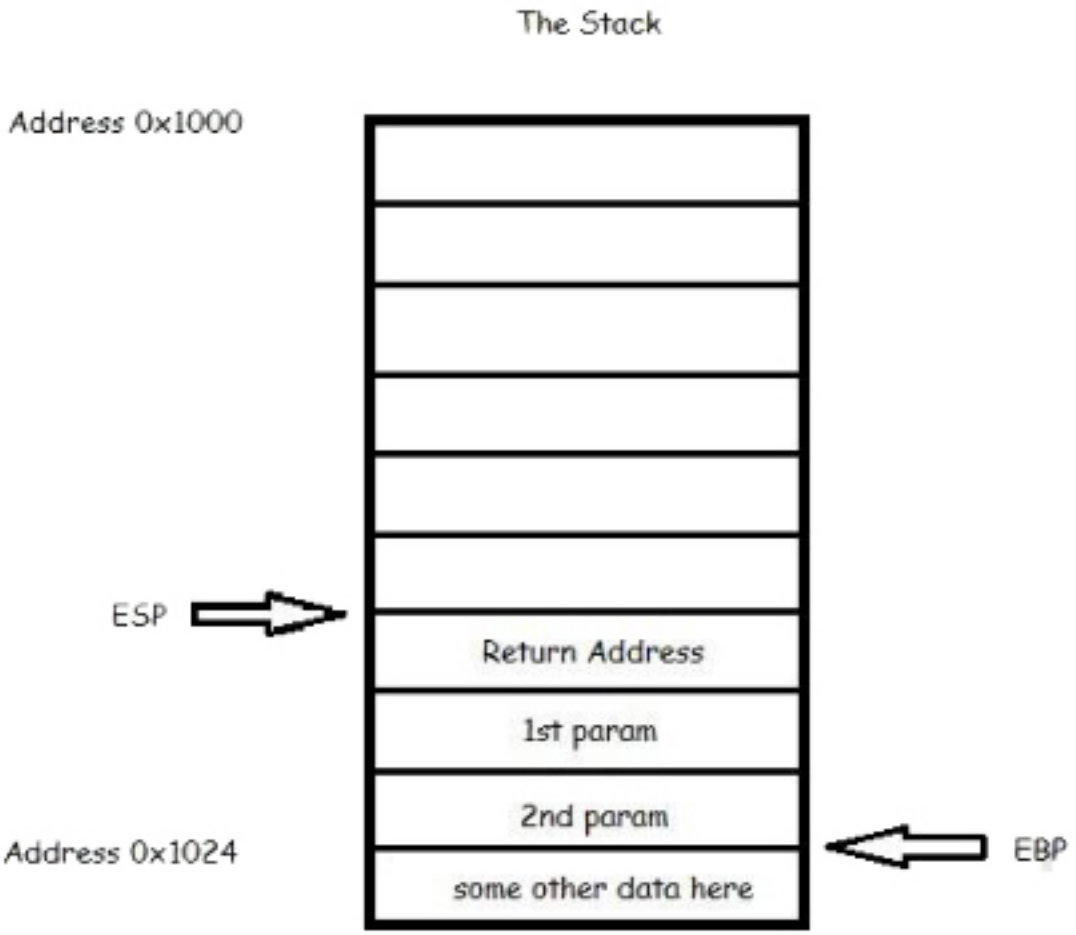


Figure 2.1 Stack Frame-1

We will now go through the stack frame setup, demonstrating the state of the stack after the execution of every instruction.

PUSH EBP

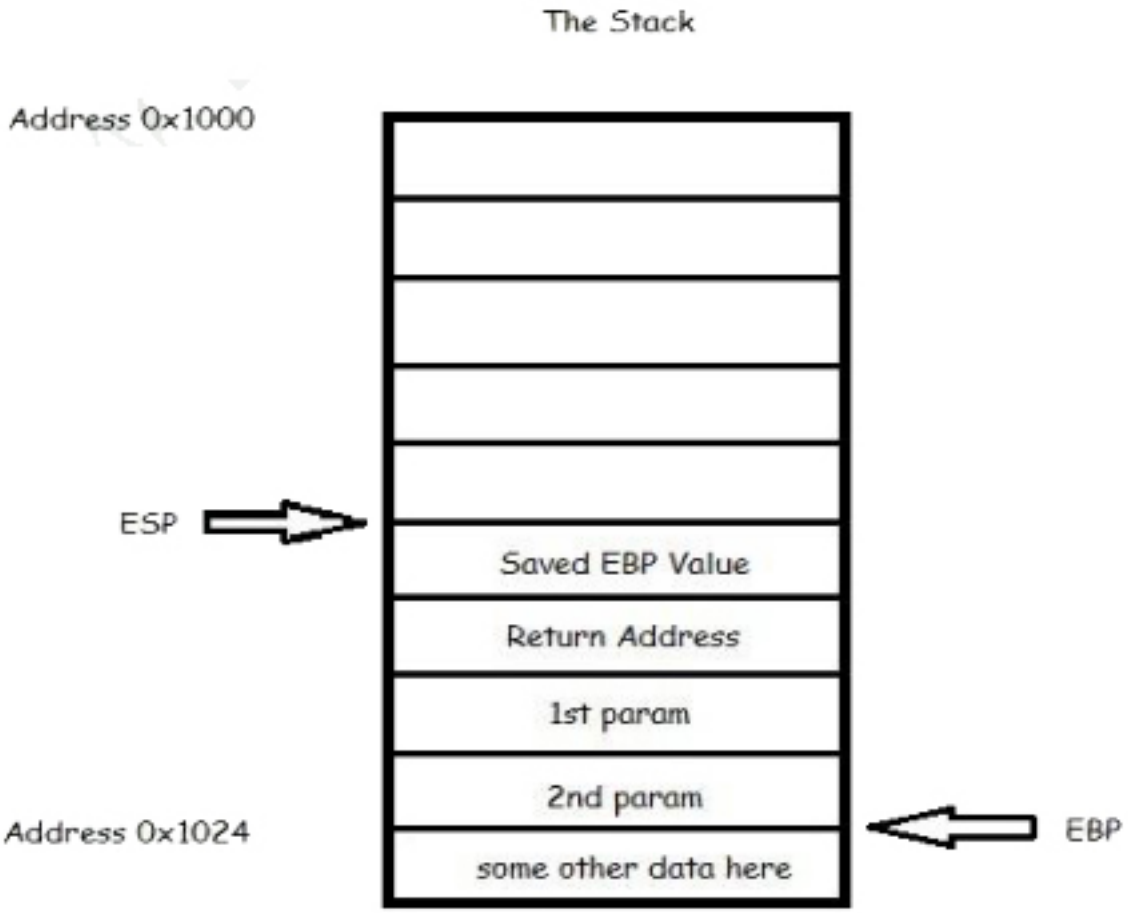


Figure 2.2 Stack Frame-2

MOV EBP, ESP

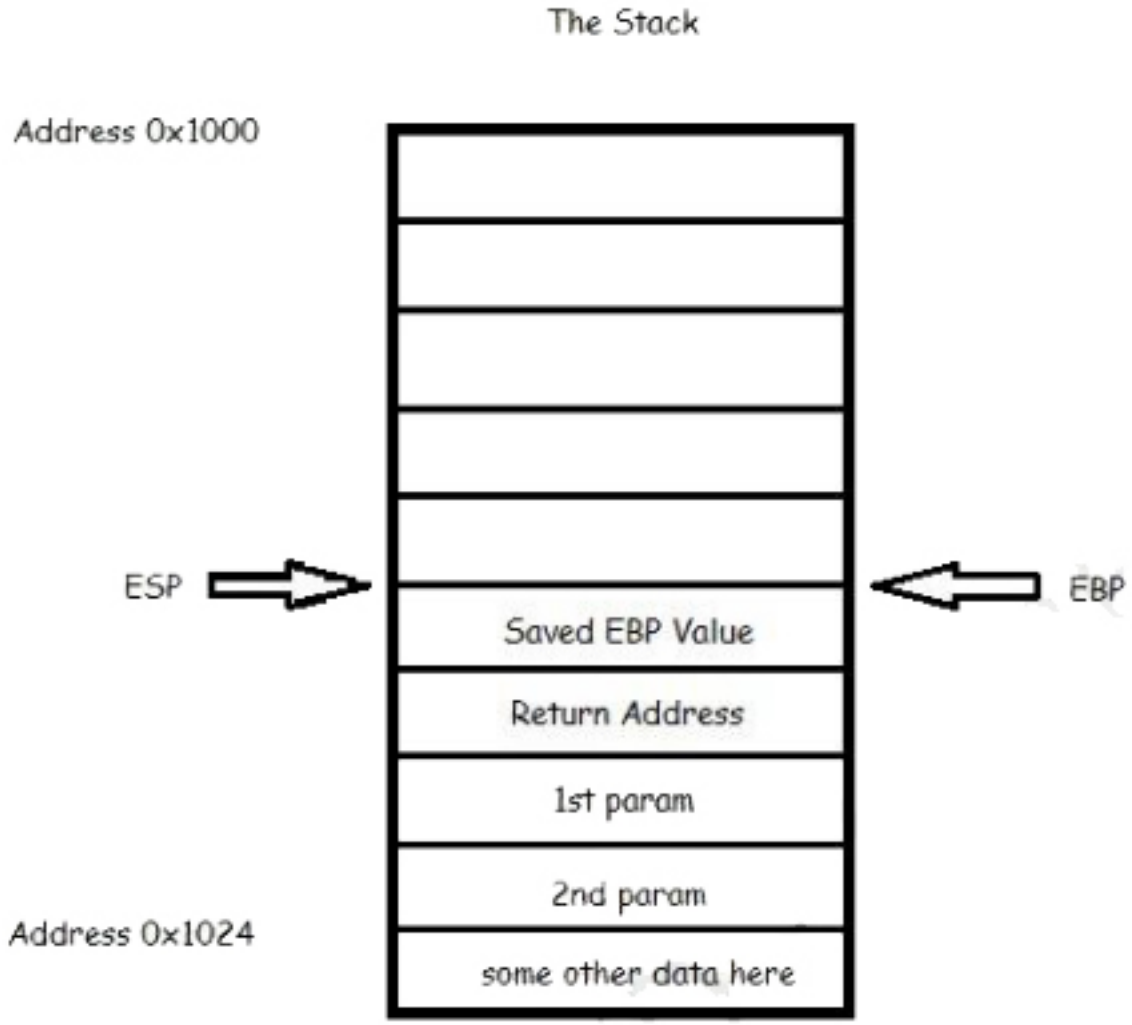


Figure 2.3 Stack Frame-3

SUB ESP, 10h

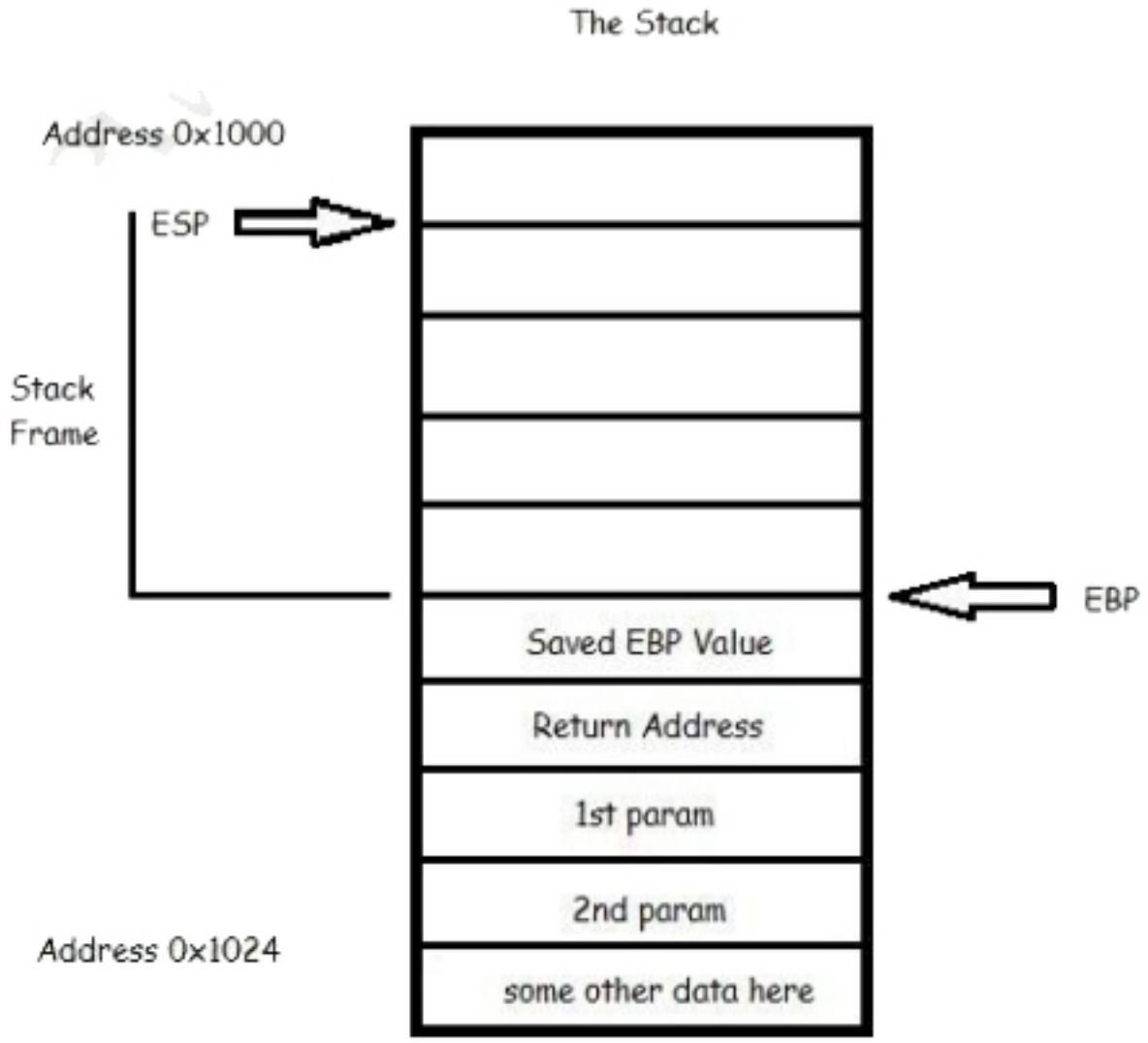


Figure 2.4 Stack Frame-4

# Calling Conventions [3]

The term calling conventions refers to the way the parameters necessary for a function are pushed onto the stack in order and the way the stack is re-balanced; this occurs inside the function or from the function that called it.

The most well-known calling conventions are __cdecl, __stdcall, and __fastcall.

In the first two calling convention types parameters are pushed onto the stack in reverse order. In other words if we have a function which accepts two parameters, for example void MyFunction(int a, intb), then calling this function will be assembled as the following:

```
push b
push a
call myFunction
```

The difference between the first two calling conventions is that the __stdcall calling convention cleans up the stack by popping the arguments (parameters) from the stack inside the function itself (the caller), but the __cdecl calling convention needs to add the cleanup code to pop the arguments from the stack after the call to the function. So, in this case, the caller is responsible for cleanup.

This means that when the __stdcall calling convention is used, the executable produced is smaller in comparison with the one produced using the __cdecl calling convention because it doesn't need the stack cleanup code inserted every time the function is called.

For example, if we call the same function ten times using the __stdcall, the stack cleanup code occurs only once in our code inside the function itself, but by using the __cdecl calling convention the cleanup code would be added ten times, to each instance immediately after the CALL instruction to that function.

Finally, in the __fastcall calling convention, the stack cleanup occurs inside the callee, but some parameters, instead of being passed via the stack before entering the function, they are passed to one or more general purpose registers. Because of this, this method makes less use of the stack.

Windows APIs (Win32 API functions) are using the __stdcall calling convention. This is because the size of the code produced is smaller, as mentioned above, and also because the amount of memory necessary for the arguments of each call is predefined, the function knows how much space to clean up before exiting

Examples

__stdcall/__fastcall (let's assume that for the __fastcall case 2 of the parameters, a and b, where pushed on the stack):

```
push a
push b
Call MyFunction
test eax,eax
JE _somewhere
```

From inside the MyFunction at the function epilogue we would see the following lines of code:

```
mov esp,ebp
pop ebp
ret 8 // lcean up the stack from parameters a and b
```

__cdecl:

```
push a
push b
Call MyFunction
add esp,8 // in this case the caller cleans up the stack
test eax,eax
JE _somewhere
```

From inside the MyFunction at the function epilogue we would see the following lines of code:

```
mov esp,ebp
pop ebp
ret // the caller will pop the parameter from the stack, so we just 'RET'
```

# Reading EIP - A simple trick

As discussed in the 1st chapter, we don't have direct access to the EIP register, but we might want to know the address of the next instruction to be executed. What we can do is create a function that returns, in EAX (or another register that you can directly access) the value of EIP.

Example:

```
GetEIP proc
mov eax, dword ptr ss:[esp]
ret
GetEIP endp
```

Or:

```
Call _getEIP
_getEIP: pop eax
```

What we can do in the first case is actually call the GetEIP function, which will move to EAX the return address, which is the next instruction after the 'call GetEIP' instruction.

In the next case we just 'call' the next instruction and pop to eax the return address, which will put in eax the address of the previous instruction executed (pop eax) that we can use to locate other parts of our code in memory, by simply adding or subtracting a value from the result.

This trick might not look useful to you, but it is widely used by 'relocatable code'. This is code (ASM) that you can inject anywhere into your source or in memory, just like shellcode is used by exploits and malicious code injection is used by malware in many cases.

# Conclusions

In this chapter we discussed processes and threadds and we also explained in detail the steps that occur during a function call, as well as the main mechanism used to allocate stack frames in order to store local variables.

We also discussed the differences between some common calling conventions and we finally demonstrated how it is possible to 'read' the value of the instruction pointer (EIP).

In the next chapter we will discuss heaps, exceptions, Windows APIs with some Windows Internals, and the most common types of reversing tools used today.

# References


1. Processes and Threads https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx
2. Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C.
3. Calling Conventions https://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx